Abstract

This document is an extension of the User's Manual and the inline documentation generated from the FWEB source files. Its objective is to provide additional details on critical aspects of the code's operation that needed by future developers.

Developer's Guide for DEGAS 2

October 26, 2020

Chapter 1

Geometry

Chapter 2

Scoring

2.1 Introduction

The estimators in DEGAS 2 are calculated by the routines in `score.web`, which are invoked during tracking. At the end of each flight, the resulting data are accumulated into the global arrays described in the statistics class (package abbreviation `sa`) via the subroutines in `stat.web`.

The data accumulation process takes place over a range of scales to optimize performance on massively parallel platforms. The basic breakdown is:

flt for "flight". This always refers to a single flight.

frag for "fragment". To facilitate load balancing, only a fraction (or "fragment") of the total number of flights per processor are sent at a time to each process. As each process completes, an additional fragment is sent until all have been completed.

fin for "final". In the simplest case, this refers to the global accumulation of data from all processes. Again, in the simplest case, the data are actually accumulated directly in an array from the output class.

This process is complicated by two additional optimizations. First, as is documented in the statistics class, these arrays may be compressed, especially at the `flt` level, to reduce the amount of data transmitted. Second, in runs on larger machines (say, > 100 cores or processes) fragments are accumulated into intermediate "final" arrays over groups of processors to reduce the number of accumulations that must be performed by the master process, avoiding network transmission collisions. The machinery facilitating this is contained in `degasinit.web`.

2.2 Basic Expressions

The sources of neutral particles to be simulated in a given DEGAS 2 run are broken up into "groups" according to their physical nature, e.g., "plate recycling", "gas puff", etc., as described in the sources class. Because the number of flights and total particle current associated with each source group are specified independently by the user, the data for each group are accumulated separately and are available by themselves within the output arrays.

The fundamental expressions used in the data accumulation process are, thus, for a specific source group i. The other principal indexing parameters for the output arrays are the geometric zones, z, and the tallies, as contained in the tally class. To minimize confusion, we do not introduce any explicit index for the latter; the following expressions apply independently for each tally. The first accumulated moment, to be associated with the mean in the end is:

$$S_1^i(z) = \sum_{j=1}^{N_i} \rho_j \left(\sum_{k \in z} \xi_{j,k} \right) / \sum_{j=1}^{N_i} \rho_j, \tag{2.1}$$

where: i is the source group index, the "1" refers to the first statistical moment, j runs over all of the N_i flights in group i, ρ_j is the relative statistical weight of flight j, k is the list of events scored by flight j (e.g., collisions) in zone z, and $\xi_{j,k}$ is the estimator value (see below) for this tally recorded by flight j during event k. The statistical weight $\rho_j \equiv 1$ in most cases, but can be set otherwise by the user to effect importance sampling via the input arrays `source_segment_rel_wt`. Note that this quantity is ***fixed*** for a given flight (indeed, is the same for all flights launched from that source segment) and is distinct from the statistical weight [`pt_w(fl_current(x))` in subroutine `follow`], designated by w in the estimator expressions below, which may vary during flight tracking, but *always* has an initial value of $w = 1$.

Estimators are discussed in greater generality in the User's Manual. Here, we write out the ones of greatest interest to provide adequate context for the statistical moment expressions. First, the estimator most commonly used in compiling test particle data is the track length estimator,

$$\xi_g^{\mathrm{TLE}} = w \, \frac{1 - \exp(-\sigma_i t_V)}{\sigma_i} g, \tag{2.2}$$

where t_V is the time the particle track spent in volume V, $w =$ weight at start of the time step. Note that in the limit $\sigma_i t_V \ll 1$, $[1 - \exp(-\sigma_i t_V)]/\sigma_i \to t_V$, so that

estimator scales directly with the time spent in a zone. For longer time intervals, e.g., as might be the case when other reaction rates are small, the exponential factor effectively accounts for weight reduction along the track. The function g can be a test particle attribute, such as mass, momentum, energy, or reaction-related, e.g., ion momentum source due to charge exchange. Volumetric plasma sources, such as the latter, enter the track-length estimator as averages over the background distribution.

The second type of estimator is the collision estimator,

$$\xi_g^{\mathrm{CE}} = \sum_{m_s} w_{m_s} \frac{g}{\sigma_s}, \tag{2.3}$$

where the sum is over m_s, all scattering collisions within V, and is equivalent to the $\sum_k$ in Eq. (2.1). The quantity w_{m_s} is the weight at m_sth collision. If suppressed absorption is being used, as is normally the case, the particle weight will decrease between the collisions. Again, g is a test particle attribute.

There is an analogous expression for the collision estimator associated with a particular reaction,

$$\xi_g^{\mathrm{CE}} = \sum_{m_j} w_{m_j} \frac{g}{\sigma_j}, \tag{2.4}$$

where the sum is now over m_j collisions of reaction j within V, The function g is some quantity related to the reaction and is likely something known only at collisions. One example of such a quantity would be the energy source due to H_2 dissociation (i.e., the energy exchange is determined only once the product velocities have been determined).

The equivalent expression for the second moment is:

$$S_2^i(z) = \sum_{j=1}^{N_i} \rho_j \left(\sum_{k \in z} \xi_{j,k} \right)^2 \Big/ \sum_{j=1}^{N_i} \rho_j - \left[\sum_{j=1}^{N_i} \rho_j \left(\sum_{k \in z} \xi_{j,k} \right) \Big/ \sum_{j=1}^{N_i} \rho_j \right]^2. \tag{2.5}$$

While this expression is very close to that for the variance appearing in statistics textbooks, it is not suitable for the multi-level data accumulation used by DEGAS 2. How one would break up the sum in the numerator of Eq. (2.1) into smaller sets (fragments) of flights j is clear. To do likewise for the second moment, we need to instead work with:

$$\sum_{j=1}^{N_i} \rho_j \left(\sum_{k \in z} \xi_{j,k} \right)^2 \Big/ \sum_{j=1}^{N_i} \rho_j = S_2^i(z) + \left[\sum_{j=1}^{N_i} \rho_j \left(\sum_{k \in z} \xi_{j,k} \right) \Big/ \sum_{j=1}^{N_i} \rho_j \right]^2. \tag{2.6}$$

2.3 Accumulating Data

We now motivate the specific expressions used to accumulate data from an individual flight into a fragment, and a fragment into a final total. In the interest of clarity, we introduce some simplified notation: $\sum_{j=1}^{N_i} \rho_j \to w$, $S_1^i(z) \to S_1$, and $S_2^i(z) \to S_2$. The subscript i in the following represents "incremental" data (e.g., an individual flight), b represents the "base" data (e.g., the fragment containing data from earlier flights); the result of the accumulation will be new base data, indicated with a prime. The familiar form for the first moment is:

$$S'_{1b} = (w_i S_{1i} + w_b S_{1b})/(w_i + w_b), \tag{2.7}$$

with the new base weight being:

$$w'_b = w_i + w_b. \tag{2.8}$$

But, the code actually uses an equivalent form to maximize numerical accuracy:

$$S'_{1b} = S_{1b} + (S_{1i} - S_{1b})\frac{w_i}{w_i + w_b}. \tag{2.9}$$

The analogous expression following from Eq. (2.6) is:

$$(S'_{2b} + S'^2_{1b})w'_b = (S_{2b} + S^2_{1b})w_b + (S_{2i} + S^2_{1i})w_i. \tag{2.10}$$

Thus,

$$S'_{2b} = [(S_{2b} + S^2_{1b})w_b + (S_{2i} + S^2_{1i})w_i - S'^2_{1b}w'_b]/w'_b. \tag{2.11}$$

The equivalent form that is coded up in `stat.web` is:

$$S'_{2b} = [S_{2b}w_b + S_{2i}w_i + (S_{1i} - S_{1b})(S_{1i} - S'_{1b})w_i]/w'_b. \tag{2.12}$$

The equivalence of Eqs. (2.11) and (2.12) can be demonstrated most readily by adding and subtracting $S_{1b}(S_{1i} - S'_{1b})w_i$ inside the square brackets of Eq. (2.11) and using Eq. (2.7) for S'_{1b}. Again, this form improves numerical accuracy, e.g., in the case where $S_{1i} \simeq S_{1b}$.

To see this, consider two limiting cases. First, look at the one in which the increment is much smaller than the base value: $S_{1i} = \epsilon S_{1b}$, where $\epsilon \ll 1$. We insert this in Eqs. (2.11) and (2.12), and drop second order terms in ϵ relative to higher order terms (as if lost due to finite precision). Note that we expect $w_i \leq w_b$

in almost all cases. since the S_2 terms are the same in these equations, we work only with the S_1 terms. So, the familiar arrangement yields:

$$S_{1b}^2 w_b + S_{1i}^2 w_i - S_{1b}'^2 w_b' \tag{2.13}$$

$$\rightarrow \quad S_{1b}^2(w_b + \epsilon^2 w_i) - S_{1b}^2(w_b + \epsilon w_i)^2/(w_b + w_i) \tag{2.14}$$

$$= \quad S_{1b}^2[w_b + \epsilon^2 w_i - \frac{w_b^2}{w_i + w_b}(1 + \epsilon\frac{w_i}{w_b})^2] \tag{2.15}$$

$$\simeq \quad S_{1b}^2[w_b - \frac{w_b^2}{w_i + w_b}(1 + 2\epsilon\frac{w_i}{w_b})] \tag{2.16}$$

$$= \quad S_{1b}^2\frac{w_i w_b}{w_i + w_b}(1 - 2\epsilon). \tag{2.17}$$

The equivalent terms from Eq. (2.12):

$$(S_{1i} - S_{1b})(S_{1i} - S_{1b}')w_i \tag{2.18}$$

$$\rightarrow \quad (\epsilon - 1)S_{1b}[\epsilon S_{1b} - S_{1b}\frac{w_b}{w_i + w_b}(1 + \epsilon\frac{w_i}{w_b})] \tag{2.19}$$

$$= \quad S_{1b}^2\frac{w_i w_b}{w_i + w_b}(\epsilon - 1)^2 \tag{2.20}$$

$$\simeq \quad S_{1b}^2\frac{w_i w_b}{w_i + w_b}(1 - 2\epsilon), \tag{2.21}$$

i.e., the same as Eq. (2.17).

However, if $S_{1i} = (1 + \epsilon)S_{1b}$,

$$S_{1b}^2 w_b + S_{1i}^2 w_i - S_{1b}'^2 w_b' \tag{2.22}$$

$$\rightarrow \quad S_{1b}^2[w_b + (1 + \epsilon)^2 w_i] - S_{1b}^2(w_i + w_b)(1 + \frac{\epsilon w_i}{w_i + w_b})^2 \tag{2.23}$$

$$\simeq \quad S_{1b}^2[w_b + (1 + 2\epsilon)w_i - (w_i + w_b)(1 + \frac{2\epsilon w_i}{w_i + w_b})] \tag{2.24}$$

$$\rightarrow \quad 0! \tag{2.25}$$

But, with the modified form,

$$(S_{1i} - S_{1b})(S_{1i} - S_{1b}')w_i \tag{2.26}$$

$$\rightarrow \quad \epsilon S_{1b}[(1 + \epsilon)S_{1b} - S_{1b}(1 + \frac{\epsilon w_i}{w_i + w_b})]w_i \tag{2.27}$$

$$= \quad \epsilon^2 S_{1b}^2 w_b w_i. \tag{2.28}$$

That is, because $S_{1i} - S_{1b}$ appears in both Eqs. (2.9) and (2.12), the factors of ϵ enter via separate terms, reducing round-off errors.

2.4 Implementation in the Code

The flexibility and generality of the scoring machinery in DEGAS 2 can make relating the expressions in the previous sections to the lines in the code challenging. Virtually all of the scoring in DEGAS 2 begins in the tracking loop of subroutine `follow` with a call to one of: `score_sources`, `score_reaction`, `score_test`, or `score_sector` (which invokes `score_diagnostics`). Note that `score_reaction` and `score_sector` also serve as the starting point for processing the corresponding action indicated by its name. This was also true of `score_sources` until the option of sampling on the master processor was implemented.

For all of the scoring routines utilizing multiple estimator types (i.e., all except `score_sector` / `score_diagnostics`), the first argument is a macro constant indicating the type of estimator to be used for that call. Note that `score_sources` is also called with the post-processing estimator in subroutine `post_process_source_scores`.

The routines `score_test` and `score_reaction` compute the estimators described in Sec. 2.2. The second argument for these represents the time-like quantity in Eqs. (2.2) - (2.4). In particular, for the calls using the track length estimator, `tl_est_track`, the variable `t_fac` is

$$t_{fac} = \frac{1 - \exp(-\sigma_i t_V)}{\sigma_i}. \tag{2.29}$$

For the calls with the collision estimator, `tl_est_collision`, the second argument is either the inverse of the total scattering rate, σ_s (`other_rate` in the code), as in Eq. (2.3), or the rate of a single process, σ_j (`rate[i]` in the code), as in Eq. (2.4).

All of the argument lists for the scoring routines contain an entry representing the current flight or, in the case of `score_sector`, the entire flight stack. In the latter case, the processing of the products of plasma-material interactions (PMI) is handled in `score_sector`, altering the flight stack. This is analogous what is done by the collision estimator call to `score_reaction` except that it returns the list of products (and reagents) in the particle array, `prod`, which is explicitly processed in subroutine `follow` into modifications of the flight stack, `fl_pointer(x)` and `fl_current(x)`. In terms of scoring, the current flight provides the weight factor, w, in Eqs. (2.2) - (2.4).

All of the scoring routines also require the `estimator_factors` argument, a 1-D array with an entry for each of the tallies. This dynamic array is actually allocated in subroutine `do_flights_master` so that it can also

be used in the post-process scoring routines as well as in the scoring routines called by subroutine follow. In the simplest cases of score_sources and score_diagnostics (called by score_sector), the nominal content of each entry in estimator_factors is the weight of the current particle. In score_test and score_reaction, this is multiplied by the time-like quantity discussed above and represented locally by the variable est_fac.

The estimator_factors array also enforces the estimator dependence of each tally contained in the tally_est_test and tally_est_reaction arrays for test and reaction / source scores, respectively. These arrays provide an overall 1 (0) factor if the current estimator is (is not) being used for a particular test tally or reaction / source tally for a particular reaction or source type (see the tally class). If all of the estimator_factors for the current score type are zero, the local flag need_scores will be set to false and the remainder of the scoring process will be bypassed.

Assuming that is not the case (i.e., need_scores is true), the quantities corresponding to g in Eqs. (2.2) - (2.4) are computed; these are stored in the local array scoring_data. The length of this array is the macro constant scoring_data_max, defined as part of the tally class in tally.hweb. The nominal content of scoring_data is the set of dependent variables contained in the problem class, i.e., the macro parameters pr_var_mass, etc. A key feature of this list is that it is extended as a part of problemsetup, e.g., to handle user-specified emission lines, with the total number of entries being given by pr_var0_num. During tallysetup, the assertion that scoring_data_max $>$ pr_var0_num is checked. The specific reason that the length of scoring_data is not set to pr_var0_num instead has been forgotten!

Subroutine test_score is the simplest of the scoring routines. The principal values of the variable g, computed in subroutine test_scoring_data, are the particle's mass, momentum vector, and energy. The pr_var_xy_stress variable, mv_1v_2, is also provided for use in the Couette Flow example. Note that additional elements of the stress tensor could be compiled in the same manner, if needed for a future application.

In subroutine score_reaction, the task of filling the scoring_data array is handled by the "collision" routines. When a collision type estimator is being processed, an actual collision routine, which results in modifications to the flight stack, is called via subroutine pick_reaction. For track and post-processing estimators, the track-length versions of the reaction routines are invoked via subroutine pick_track_reaction. These have no impact on the characteristics of the current flight; in fact, subroutine follow asserts that these calls to

score_reaction return no products. All of these reaction routines provide the entries in scoring_data corresponding to the changes in "problem species" (the union of the test and background species lists) mass, momentum vector, and energy. For those processes that result in the emission of photons, e.g., ionization and dissociative excitation, values for the emission rate and velocity vector of the emitting species are included in the scoring_data array.

Each of the source types has a corresponding "data" routine, e.g., subroutine puff_data, that is called from subroutine score_sources and fills in the scoring_data mass, momentum vector, and energy sources for it. Subroutine plate_data handles both the so_plate and so_plt_e_bins types. A recombination source is handled more like a reaction, however, in that it provides scoring data for multiple species and also results in light emission. To underscore this point, the recombination_data subroutine is actually kept in the collide.web file.

2.4.1 Sector and Diagnostic Scoring

The scoring of sectors and diagnostics in score_sector is complicated first by the presence of two sectors at a zone boundary and second by the need to track mass, momentum, and energy both into and out of the sectors. The subroutine first sets sector1 to lc_sector and sector2 to lc_sector_next for the current flight. Their values will be used to determine if the current flight is at an exit, in which case the current flight is terminated immediately. Or if it is at a wall / target interface, in which case a PMI is processed. The comment after the setting of sector1 and sector2 described the objectives of the next three lines, but understanding them is crucial to comprehending the subsequent subroutine calls:

```
nprod = 0
```

Anticipates the possibility of a purely diagnostic interface in which no PMI or exit is processed; i.e., the current flight goes on its way as is.

```
pt_copy(fl_current(x),prod[0])
```

Both the process_pmi and score_diagnostic subroutines operate on prod[0] (and subsequent elements in the case of a PMI), so it must be a copy of the current flight particle.

```
pt_thru_face(fl_current(x))
```

Again, this line anticipates the possibility of a purely diagnostic interface. The macro invokes `lc_thru_face` in `location.hweb`, which in turn sets `lc_cell` to `lc_cell_next` and likewise for `lc_sector` and `lc_sector_next`. As implied by its name, the macro effectively puts the particle on the other side of the interface. If there is a PMI or the current flight is at an exit, the flight is killed, and this macro does not matter. This has no impact on the scoring since that is now going to be done on `prod[0]`.

These lines are followed by a loop over the non-diagnostic sectors; i.e., `type` $=$ `sc_vacuum` $= 1 \rightarrow$ `sc_exit` $= 5$, which is also the value of `sc_diagnostic(0)` (see the corresponding macro in the sector class). The way this loop is written amounts to "overkill" in that both "`sector`" and "`sector_next`" are tested and that only three of the five sector types (wall, target, and exit) result in any action. This structure harks back to earlier incarnations of the "sector" concept. In the wall and target cases, the requisite PMI is processed.

The operation of PMI related routines is discussed elsewhere (e.g., in the User's Manual sections on Recycling and Adding Reactions and PMI), but noting a few details here will help clarify the operation of sectors and diagnostics. Note, first, that the PMI yields may be functions of the test particle incident angle, θ, measured relative to the surface normal, and $\cos(\theta)$. The latter is provided by the function `intersection_direction`, which in turn invokes `surface_intersection_direction`. As is noted there, $\cos(\theta) > 0$ means the particle is leaving that surface and cell.

The velocity of the PMI product is returned by the PMI routines as if the particle were impinging on the $X - Y$ plane. When these routines return the product particles to subroutine `pick_pmi`, the outgoing velocity vector is rotated into the proper local orientation by subroutine `surface_reflect`. This is then loaded into the velocity of the `prod` array of each product along with the location of the incoming product. However, this only amounts to copying its location in configuration space, $\vec{x}$, since the next lines of code proceed to set the `lc_cell` to the products to the incoming `lc_cell_next` and vice versa; the same is done for the sector and zone values. This swap is explained in the comment: the scoring is performed as if the incoming (outgoing) particle is leaving (entering) the plasma or vacuum zone and entering (leaving) the solid zone. Once control returns to subroutine `score_sector`, `score_diagnostics` is called. Before delving into it, note that upon its return, the `pt_thru_face` macro is called for each product so that each is then labeled as being in a plasma or vacuum zone (and cell) and is, thus, ready to be tracked. The face and sector values at this point are incorrect, but these are not needed for tracking and test or reaction scoring.

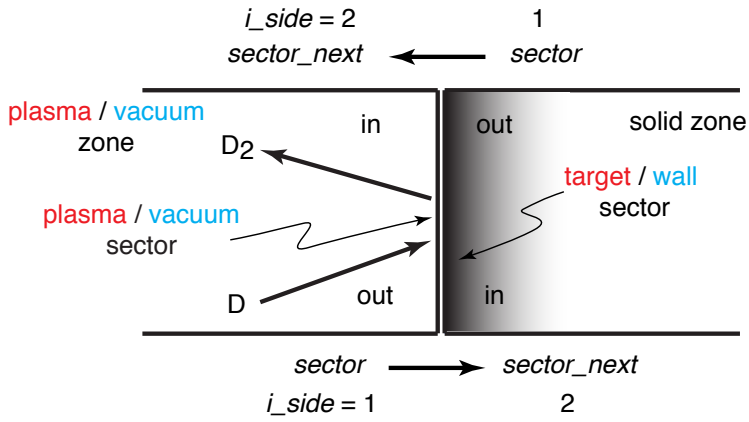

Figure 2.1: Schematic for a desorption PMI showing the sector values used by subroutine `score_diagnostics`.

The values of the `sector` and `sector_next` for the current and product particles when `score_diagnostics` is called are shown schematically in Fig. 2.1. An outer loop, `j` $= 0 \rightarrow$ `nprod`, cycles through each of the current and product particles. Inside of this is a loop over all of the tallies of type sector, with tally number `jscore`. Yet another loop with index `i_side` considers the two sides of the interface; as in Fig. 2.1, the values of the local variable `sector` is set using either `lc_sector` or `lc_sector_next`, likewise for the local variable `zone`.

The `surface` identifier corresponding to `sector` is obtained from the `sector_surface` array; this is unique. The number of sectors, `count`, associated with this surface then comes from `surface_sectors`. The value of `count` may be > 1 for two reasons. First, in 3-D cases a single surface may bound all of the zones constructed from a discretization in the toroidal direction. However, only one of those will have the same zone as the one being scored. More relevant in this case is that purely diagnostic sectors may have been defined for this surface and zone combination; see the description of auxiliary sectors in `definegeometry2d` (note that much of the above description of sectors, etc. is echoed in its preamble and other documentation).

Because of this latter possibility, yet another loop goes through each index of the loop looking at one of these sectors, sector number `subsector`. The key point of this set of nested loops is to identify which tallies need to be updated at

the particle's location. For this to be the case, three conditions must be satisfied:

1. The "geometry operator" for the tally `jscore` (e.g., "Wall and Target Counts"; see the tallysetup documentation) must include one of the sectors `subsector` associated with `sector`.

2. The zone associated with `subsector` is the same as the current `zone` value.

3. The dependent variable for tally `jscore` is consistent with the "in" / "out" convention depicted in Fig. 2.1.

If these conditions are met, `estimator_factors[jscore]` is set to the particle weight; if not, the value is left at the default value of zero. The last of these three is a key point; this is where the directionality of the diagnostics is enforced. In principle, this could have been done (and in earlier versions of the code, was done) in the subsequent subroutines `sector_scoring_data` or `handle_diagnostics`.

At the completion of these loops, but still inside the loop over the current and PMI product particles, `sector_scoring_data` is called to fill in the `scoring_data` entries for angle, energy, as well as the mass, momentum vector, and energy in both the "in" and "out" directions. These are actually the same since, again, the directionality is enforced in subroutine `score_diagnostics`.

2.4.2 Subroutine `add_scores`

At the end of each of the main scoring routines, such as `score_test`, is a call to subroutine `add_scores`. This is the point at which the information from this step (`score_test`), collision (`score_reaction`), sector crossing (`score_sector`), or source sampling (`score_sources`) gets added to the `stat_flt` array that contains all of the scores generated by the current flight. Each invocation of `add_scores` is, thus, for a particular type of tally: sector, test, or reaction; this is the first argument for `add_scores`. The second argument is a particle; this will be the current particle in the case of `score_test`, `score_reaction`, and `score_sources`. In `score_diagnostics`, `add_scores` is called separately for each PMI product (if any; purely diagnostic sectors have no products with `nprod` $= 0$). The third argument is the problem reaction number; this is non-trivial only for the calls by `score_reaction` and `score_sources`.

The index_parameters array contains the values of the tallys' independent variables. The corresponding indices are set in the tally class; see also subroutine set_var_list in tallysetup.web.

The main loop in the subroutine is over all tallies, jscore, of the type specified by the first argument. The first task performed in this loop is a check to see if "track conversions" (see tally class documentation and tallysetup) are needed for jscore. Next comes a test to see whether or not the dependent variable for the score jscore is one of the mass, momentum, or energy change variables (from a "collision" reaction or a PMI sector). If it is, the start and end points for a loop over problem species are defined, otherwise, these are set to trivial values. In both cases, the check_data variable is compiled from the corresponding entries in the scoring_data array to determine whether or not its contents are trivial. If the latter or if the estimator_factors value for jscore is zero, the current loop is exited, potentially saving significant effort.

Should both the estimator_factors and check_data values be non-trivial, the geometric aspects of the score are sorted out by subroutine handle_geometry (see tally class documentation. For volume tallies, the value of number_segments is 1 on return and the content of the segments array is just the zone number of the current particle. For reaction (sector) tallies, handle_detectors (handle_diagnostics) is invoked to fill in these data, as well as any associated "binned" data.

For detectors, the list of "segments" consists of the detector chords associated with the tally jscore (its "geometry operator" or "geometry ponter") that have non-zero contributions from the current zone. If the emission spectrum is being computed for the tally, the "bins" contain that information. Additional details on this can be found elsewhere.

The analogous steps for the sector based diagnostics are taken in handle_diagnostics. The structure of this subroutine resembles that of score_diagnostics in that it again examines the sectors on the two sides of the current interface. As noted above, the number of sectors to be scored on one of those sides may be > 1 due to the specification of a non-default diagnostic surface there. If no non-default diagnostic surfaces have been defined, the returned value of number_segments $= 1$. Moreover, the associated sector will be part of either a solid or exit zone since that's the case for all default sectors. One difference between score_diagnostics and handle_diagnostics is that the latter is focused on a single diagnostic group, like "Wall and Target Counts"; the list of "segments" returned is of members of that group.

Following the call to handle_geometry, add_scores loops over these

segments and their associated "bins", if any, and sets a corresponding entry in the index_parameters array. A third loop goes through all of the "problem species", if needed for this score, and setting the entry for each in index_parameters; if not needed, the loop is trivial. One final loop then goes through each of independent variables for the score jscore and sets its entry in ind_val from the contents of index_parameters. Once these are all set, and, if needed, "track conversions" applied, set_comp_scores is finally called to update the local copy of the output arrays.

2.4.3 Subroutine set_comp_scores

The subroutine itself is relatively simple. Only two arguments are needed: the new contribution to the current score and its location in the output array. The latter is provided by the out_array_index macro (see output class documentation), which is constructed using tally_base, tally_dep_var_dim, and tally_tab_index. The new contribution, datum, is the relevant entry in the scoring_data array multiplied by the geom_mult value returned by handle_geometry (generally is $= 1$ except for detector based scores) and the value of estimator_factors for jscore. The seemingly complex sequence of macros needed to identify the entry in scoring_data handles the dependence on problem species and on the rank of the dependent variable (for vector quantities, such as "momentum change").

For the case in which stat_comp_flt is false, set_comp_scores, updates the relevant entry in the stat_flt array and returns. If compression at the flight level is turned on, the default, the subroutine first checks to see if an entry in the ifull slot of the output arrays has been made yet. If so, that entry is updated using the stat_ptr2short_flt mapping (see the stat class documentation). If not, the size of stat_flt array is incremented, the array reallocated if needed, and the pointer arrays stat_ptr2short_flt and stat_ptr2full_flt assigned.

The end result of all of the calls to score_test, etc. for this flight as it is tracked through subroutine follow is effectively the quantity S_1 discussed in Sec. 2.3.

2.4.4 Post Processed Scores

For reaction type tallies, the user has the option of scoring specific reactions after all of the flights have been run and all of the data have been accumulated. This pro-

cess is initiated in a section of code near the end of subroutine `do_flights_master` (in `doflights.web`). The code first checks to see if any of the reaction type tallies use the "post processing" estimator, `tl_est_post_process`; if so, the logical variable `need_scores` is set to `true`. This is followed by a loop over source groups. The "post" and "output" final conversions (see tally class documentation and tallysetup documentation) are applied in this loop; these are performed even if `need_scores` is `false`.

If `need_scores` is `true`, the subroutines `post_process_test_scores` and `post_process_source_scores` are called for each source group. The only arguments to the subroutines are the source group number and the same `estimator_factors` array introduced earlier in Sec. 2.4.

The basic idea behind `post_process_test_scores` is that we create a fake particle (i.e., an instance of the particle class) having the location of each zone in the problem space and weight equal to the number of particles in that zone accumulated during the main run. Because the plasma parameters are constant throughout that zone, all of the reaction rates are the same for these particles. For some scores, such as ionization and light emission rates, the resulting tallies will be the same as those obtained with a track length estimator.

In the original implementation, the "divide number" conversion was used to transform the neutral flux into an average particle velocity that could be used in this same way to obtain post-processed tallies of momentum sources. However, that conversion was found to give incorrect results in runs with multiple source groups. For this reason, an arbitrary velocity vector is used, rendering these post-process tallies incorrect. The same is true for energy source tallies. Note that even if "divide number" worked as intended, additional code would be required to correctly post-process energy source tallies since the particle class does not have an "energy" property that could be used to transmit the average particle energy in the cell to the reaction scoring routines.

Post-processing is nonetheless useful for some applications. One example is the "fixed neutral density" approach to computed a synthetic GPI diagnostic. That fixed neutral density is obtained via a normal run of the code, using a track-length estimator with a particular plasma background. Subsequent plasma time steps are then run as restarts of DEGAS 2 based on the output file from that initial run, but having the same number of flights. Since the number of flights is unchanged, the main loop of the code does nothing. By changing the scoring of the reactions for the light emission related tallies from track-length or collision to post-process, the light emission rates are re-evaluated with the new plasma parameters, effectively using the same neutral density. These restart simulations take vastly much less

time than the original.

The actual implementation in post_process_test_scores is, in hindsight, needlessly general. For example, the particle number (and neutral flux vector) tally has had only zone number and test species as the independent variables in virtually every known run of the code so that the explicit loops over all five (tl_rank_max) possible independent variables is overkill. The only other less-than-obvious aspect of the code is indicated by the existing comment noting that using the lc_set_a macro to set the particle location is potentially time consuming and unnecessary.

The post-processing of source scores is more straightforward in that they are based on the underlying kinetic distributions. Note that a post-processing option has not yet been set up for snapshot sources. Doing so is possible, in principle, and may be advantageous in that the resulting sampling of the snapshot distribution might be more thorough (and, thus, more accurate) than that obtained at run time. Subroutine post_process_source_scores loops over all segments, seg, of the current source group. The corresponding kseg = seg -1 and xseg (pointer to a geometry element, such as a sector or zone) parameters are, thus, easily obtained from the segment number. Just as with a post-processed test score, a fake particle is defined for the scoring process with weight equal to the source current at segment seg. The fake particle's location is intended to approximate the average obtained by set_source_x in run time sampling. For surface sources, the midpoint of the line segment is used; for volume sources, it's the zone center. The fake particle's velocity is non-zero only in the case of a recombination source. In the other cases, the source sampling process does not depend on this velocity so that it can be safely set to zero. The fake particle is then turned into a fake flight and passed on to the score_sources subroutine.

This subroutine works in exactly the same way as during run time. The distinction arises only in the subroutine calls that set up the "data" for each source type; in each case, the values of sampled parameters are replaced by their averages. For example, in the case of a plate source (subroutine plate_data), the parameter wa $= 3$; in run time, this is the square of the unscaled velocity vector obtained by sampling from a Gaussian distribution in each of the three spatial directions (via rn_gauss_next). The parameter vpuff serves this exact same purpose in subroutine puff_data. In this case, one also needs the average velocity vector as sampled from a cosine distribution, which is $(0, 0, 8/3\sqrt{6\pi})$. Note that this is correct only in the default case of a pure cosine distribution; the more general "cosine to an exponent" case is not handled. For a plate source with a specified energy distribution, subroutine plt_e_bins_v_details uses the input param-

eter wa (in call from plate_data) / urd (its argument list) as a switch, with a value of the macro real_unused telling it to compute the average energy from the user specified energy distribution.

For the volume source, the user specifies the source's average velocity, $\vec{v}_{\rm src}$, and temperature, T. The former becomes the fake particle's velocity. Its energy is then just $1.5T + 0.5mv_{\rm src}^2$.

Scoring the recombination source is more complex due to the associated light emission and electron energy losses and sinks. The fake particle's velocity is set from the recombining ion's flow velocity in post_process_source_scores. In subroutine recombination_data, the electron energy source and loss rates are evaluated, as are the wavelength and emission rate of the diagnostic lines being simulated. The ion energy lost is the thermal value analogous to that shown above for the volume source. The flow velocity vector and magnitude are loaded into dedicated "Maxwellian emitter" slots in the scoring_data array. When add_scores is called at the end of score_sources and it in turn calls handle_detectors, these data are used to compute the corresponding full emission spectrum. This provides an enormous improvement in signal quality over a collision-based score, making this another very useful application of post-processed scoring.

2.5 Final Processing

The expressions in Sec. 2.3 above are incorporated into subroutine stat_acc, although their equivalence is difficult to discern due to the bookkeeping associated with the compression of the S arrays. This routine is called at various points during the run to first accumulate "flight" data into a "fragment" array, and then the "fragments" into the "final" array. The end result of this process is the output_grp array via the call to stat_acc in the "Merge results" macro invoked by subroutine do_flights_master.

As will be shown in more detail below, the mean tally values for each source group are scaled by the total current in that group, which is obtained by summing over the group's $N_{{\rm seg},i}$ segments:

$$\Gamma_i = \sum_{\ell=1}^{N_{{\rm seg},i}} \Gamma_{i,\ell}. \tag{2.30}$$

In the steady state mode of operation, Γ_i represents a number of particles per second; in the time dependent mode, Γ_i is a number of particles. Obviously, these

quantities are used in combining the $S_1^i(z)$ and $S_2^i(z)$ over source groups. But, they are also used to scale the user-specified relative statistical weights to get the ρ_j appearing in Eq. (2.1), etc. above and in setting the probabilities for sampling flights from the source segments.

Namely, the user can specify an arbitrary set of relative weights, $r_{i,\ell}$, for segments $\ell = 1 \to N_{\mathrm{seg},i}$ in group i. A normalization factor is computed for each source group:

$$r_{\mathrm{norm},i} = \left(\sum_{\ell=1}^{N_{\mathrm{seg},i}} \Gamma_{i,\ell}/r_{i,\ell} \right) / \left(\sum_{\ell=1}^{N_{\mathrm{seg},i}} \Gamma_{i,\ell} \right). \tag{2.31}$$

The cumulative probability distribution used in sampling flights in group i is, e.g., for segment ℓ:

$$p_{i,\ell} = \left(\sum_{m=1}^{\ell-1} \Gamma_{i,m}/r_{i,\ell} \right) / \left(\sum_{m=1}^{N_{\mathrm{seg},i}} \Gamma_{i,m}/r_{i,\ell} \right). \tag{2.32}$$

These probabilities are computed without the weight normalization factor Eq. (2.31) since it would appear in both numerator and denominator. The actual sampling procedure is more complex and offers additional flexibility; see the sources class and sourcetest.

If flight j in group i is sampled from segment i, the relative weight is set to:

$$\rho_j = r_{i,\ell} \times r_{\mathrm{norm},i}. \tag{2.33}$$

Scaling and final processing of the output arrays is carried out by a couple of loops and subroutine calls near the end of subroutine `do_flights_master`. In the first of the loops, the means and variances are scaled by the relative source currents:

$$P_1^i(z) = \frac{\Gamma_i}{\sum_i \Gamma_i} S_1^i(z) \tag{2.34}$$

and

$$P_2^i(z) = \left(\frac{\Gamma_i}{\sum_i \Gamma_i} \right)^2 \frac{S_2^i(z)}{\left(\sum_{j=1}^{N_i} \rho_j - 1 \right)}. \tag{2.35}$$

At this point, S represents `output_grp`, and P represents `out_post_grp`; the latter is also summed over source groups to obtain `output_all`.

A second set of loops performs a final scaling by the global total current, $\sum_i \Gamma_i$, and a division by the time interval for time dependent runs. The latter transforms the input "current" from a number of particles into an average number of particles

per second; in the interest of clarity we do not explicitly show this factor here. At this stage, the variance is transformed into the relative standard deviation that is used externally to quantify the accuracy of the code's results:

$$P_2'^i = \frac{(P_2^i)^{1/2}}{P_1^1}, \tag{2.36}$$

and

$$P_1'^i = (\sum_i \Gamma_i) S_1^i \tag{2.37}$$

Namely, the expected error in a statistical estimate of the mean μ via N samples is $\sigma/\sqrt{N}$, where σ is the standard deviation and σ^2 is the variance. The relative standard deviation "rsd" is then

$$\mathrm{rsd} = \frac{\sigma}{\mu\sqrt{N}}. \tag{2.38}$$

Note that Eq. (2.5) and so on have $N \leftrightarrow \sum \rho_j$ in the denominator of the variance even though $N-1$ appears in most textbook expressions. The "-1" accounts for the fact that the same sample is used to estimate the mean μ and the variance. While $N \gg 1$ in most DEGAS 2 runs, rendering the difference between N and $N-1$ negligible, retaining $N-1$ yields a maximum relative standard deviation of exactly unity when $\rho_j \equiv 1$ (a single score; see below), facilitating interpretation of results. In the process of computing the relative standard deviation, we effectively multiply the S_2 by $N/(N-1)$. The $N-1$ part of this is the denominator in Eq. (2.35); the N ends up being cancelled by the $1/\sqrt{N}$ in Eq. (2.38) and does not explicitly appear in Eq. (2.40).

The $P_1'^i$ and $P_2'^i$ quantities in Eqs. (2.40) and (2.39) again end up in the `out_post_grp` array and the sum over source groups in `output_all`.

For purposes of explication, we combine the expressions in Eqs. (2.34 - (2.39) into:

$$P_1'^i = \Gamma_i S_1^i, \tag{2.39}$$

and

$$P_2'^i = \frac{\left[\Gamma_i^2 S_2^i / \left(\sum_{j=1}^{N_i} \rho_j - 1\right)\right]^{1/2}}{\Gamma_i S_1^i}. \tag{2.40}$$

Finally, post-processed tallies are added to the `out_post_grp` array and the "conversion" factors are applied via calls to subroutine `final_conversion`. The mean values in this are array are summed over groups to obtain the means in `out_post_all`. Since the effects of the post-processing tallies on the variance

are *not* accounted for, the rsd's from `output_all` are copied into the "variance" entry in `out_post_all`.

2.6 External Use of Output Arrays

The principal output of the code is contained in the mean values of the `out_post_grp` and `out_post_all` arrays. Invariably, the relative standard deviations in these arrays are used to characterize the uncertainty in the results, although the inclusion of post-processed tallies may render those values inaccurate. To be safe, one should use the rsd's from "test" tallies, which are not available for post-processed scoring, to quantify the uncertainty in a simulation.

None of the scaling factors and transformations described in Sec. 2.5 are ever applied to the `output_grp` array since it is used as input to restarted simulations.

This fact is forms the basis for the `acc_output` utility, which is used to effectively average over a number of simulations. A related application is to simulate neutral transport in the presence of a time varying plasma background (since the background is always assumed fixed in time during a given run of DEGAS 2).

The `acc_output` utility is called after each run with the name of a "base" output netCDF file specified as a command line argument. On the first such call, this file is assumed to not exist. The output file specified in the `degas2.in` file is read in and its `output_grp` array is scaled by Γ_i for the mean and Γ_i^2 for the variance. These source strengths are read in from the corresponding background file. The resulting array is written into the `output_grp` array in the "base" file. Other data appear in the "base" file at this point, although they are not used.

On subsequent calls, the `output_grp` data in the `degas2.in` output file are again read in and scaled by the source strengths. The latter come from the current background netCDF file; this is an important consideration in that they may differ from those of the previous run in a time dependent application. At this point, the code invokes expressions Eq. (2.8), (2.9), and (2.12), just as in the `stat_acc` routine; these results are placed in the `output_grp` routine of the "base" file, facilitating successive repetitions of this process.

With these data now representing a non-trivial product of the input data, expressions equivalent to Eqs. (2.39) and (2.40) are applied and inserted into `out_post_grp`; these are summed over groups to fill `output_all`. By design, the Γ_i factors have already been applied. Because the background plasma data may be different in the constituent simulations, post-processed tallies cannot be handled; the code will throw an assertion if any are present. The conversions performed by

subroutine `final_conversions` *are* applied, yielding `out_post_grp` and `out_post_all`. Due to the exclusion of post-processed tallies, the variances of both sets of output will be accurate. The resulting "base" file can then be used just like a normal output file in conjunction with `outputbrowser`, `geomtesta`, etc. The two are not completely equivalent since `acc_output` scales the `output_grp` array and does not update the `output_2D_coupling` array (those data can be obtained directly from the constituent tallies via `outputbrowser`). The arrays `output_num_flights` and `output_random_seed` are not utilized, either. The total, accumulated weight *is* available in `output_weight_grp`.

2.7 A Single Score

In the special case of a single score ξ_x with weight ρ_x in a particular zone x, Eq. (2.1) simplifies to

$$S_1 = \frac{\rho_x \xi_x}{\rho_{\rm tot}}, \tag{2.41}$$

with $\rho_{\rm tot} \equiv \sum_j \rho_j$. Equation (2.5) becomes

$$\begin{aligned} S_2 &= \frac{\rho_x \xi_x^2}{\rho_{\rm tot}} - \frac{\rho_x^2 \xi_x^2}{\rho_{\rm tot}^2} && (2.42) \\ &= \xi_x^2 \frac{\rho_x}{\rho_{\rm tot}} \left(\frac{\rho_{\rm tot} - \rho_x}{\rho_{\rm tot}} \right) && (2.43) \end{aligned}$$

Then, the relative standard deviation, which is effectively $\sqrt{S_2}/[\sqrt{(\rho_{\rm tot}-1)}S_1]$ reduces to:

$$\mathrm{rsd} = \frac{1}{\sqrt{\rho_x}} \left(\frac{1 - \rho_x/\rho_{\rm tot}}{1 - 1/\rho_{\rm tot}} \right)^{1/2}. \tag{2.44}$$

In the limit of $\rho_{\rm tot} \gg 1, \rho_x$, the $\mathrm{rsd} \rightarrow 1/\sqrt{\rho_x}$. In the usual case of uniform weighting, $\rho_x = 1$ and $\rho_{\rm tot} \rightarrow N$, $\mathrm{rsd} \equiv 1$, as noted above.

Note, however, that when performing importance sampling, one can obtain single scores having $\rho_x < 1$ and, thus, an rsd > 1. This is simply a reflection of that score carrying less than average statistical weight. The rsd values in regions of the problem space having adequate statistics can be interpreted via the same guidelines as simulations with uniform weighting (e.g., as in Sec. 2.4 of the User's Manual).

2.8 Handling of Relative Weights for Importance Sampling

The path of the relative weight factors through the code, particularly in the time dependent case, is sufficiently complex that it warrants exposition.

2.8.1 Setting up the source (e.g., in `defineback`)

First, the relative weights are stored in the array `source_segment_rel_wt`. The default value of 1 is set in subroutine `init_wt_alias` in `writeback.web`. Since importance sampling is rarely used, and schemes for employing it must be adapted carefully to the problem at hand, no specific code for it is provided; the user is responsible for inserting code that will assign the desired values.

As an example, we provide a rough description of its application to the time dependent version of the synthetic Gas Puff Imaging diagnostic, where the objective is to ensure radially uniform sampling of particles from the snapshot source. For simplicity, the relative weight computation is integrated into subroutine `set_snapshot_source`; the lines establishing the default values in subroutine `init_wt_alias` are then commented out. In `set_snapshot_source`, the problem space is divided up, somewhat arbitrarily, into 1 cm radial bins, and the weights of the snapshot particles are accumulated into them. The total weight in each bin is then divided by the global total to yield a fraction; this becomes the relative weight factor associated with each bin and, thus, each snapshot particle in that bin.

As in the default version of subroutine `set_snapshot_source`, the `source_current` for each particle (i.e., each particle is a "source segment") is set to the snapshot particle's weight (without the relative weighting factor) and the total current, `so_tot_curr` for the source group. The relative weight factors are then combined with the currents in Eq. (2.32) to set up the sampling distribution. The idea is that bins deeper into the plasma, where penetration is poor, will have smaller relative weights. This will cause them to be preferentially sampled, as in Eq. (2.32). An unbiased result is achieved by reducing the statistical weight of these particles in the main code by the same relative weight factor. The overall weight normalization factor, `so_wt_norm` is computed (by default) from the currents and relative weights via Eq. (2.31) in subroutine `set_prob_alias`.

2.8.2 Source sampling

As is noted in sources.web, subroutine sample_sources, the value of source_segment_rel_wt, scaled by so_wt_norm, is assigned *temporarily* to the sampled particle's weight, pt_w. In the ff_copy macro (flight_frag.hweb) invoked in doflights.web, subroutine setup_flight_frag,this scaled relative weight is transferred into the ff_float_pt_w entry in the ff_particles_float array.

The associated ff_label_init macro, also in flight_frag.hweb, returns this quantity as a macro parameter, rel_wt, and then sets the flight's (instantaneous) statistical weight, pt_w, to 1. The overall weight of the flight, stat_wt_tot_flt, is equated to that rel_wt parameter with the invocation of the macro in flight.web, subroutine do_flights; it is then globally accessible via the "stat" (sa) class and associated common blocks. This is the weight that is passed to subroutine stat_acc to accumulate all of the scores from this flight and to obtain the total sampled weight for the source group. That is, the individual scoring routines, such as score_test, record only pt_w. However, when all of those scores are combined with those from preceding flights, they are effectively scaled by the relative weight factor via stat_wt_tot_flt. Note that with importance sampling the total weight in the source group is only approximately equal to the number of flights sampled.

2.8.3 Snapshot score and PDF

Flights (in any source group) reaching the end of the time step in time dependent runs trigger a final call to score_test with the snapshot estimator, tl_est_snaphshot. This will, obviously, lead to the accumulation of data needed for tallying the snapshot density, etc. But, it also results in a call to subroutine build_snapshot_pdf in score.web. The weight of the resulting entry in the snapshot particle distribution function, sn_float_pt_w, is recorded there as the product of pt_w and stat_wt_tot_flt. Once all of the particles in a source group have been tracked, the weights of the newly created snapshot particles are scaled by the total current in the group, so_tot_curr (and the rarely used, arbitrary factor so_scale), and divided by the total weight in the group, output_weigh_grp. As noted above, in the subsequent time step, subroutine set_snapshot_source (writeback.web) sets the source_current associated with this particle to the value of this sn_float_pt_w entry in the snapshot arrays.

Chapter 3

Synthetic GPI Example

3.1 Introduction

This document guides the user through an example synthetic gas puff imaging (GPI) simulation using XGC data. Additional details are provided to facilitate future applications.

3.2 Geometry Setup

3.2.1 `efit2dg2d`

The triangular mesh used by XGC is too fine for this application, resulting in too long run times and / or excessive Monte Carlo noise levels, as well as stressing memory and storage resources. We instead use the `efit2dg2d` routine to set up a triangular mesh over the volume of interest.

The linear dimensions of the XGC mesh triangles are roughly 1 mm. The parameters in `efit2dg2d` and subsequently in `definegeometry2d` are set so as to result in triangles roughly three times this size, significantly reducing computational requirements, but still adequately resolving spatial variations in the XGC plasma data. Namely,

1. The box size and number of flux surface contours are chosen such that the average radial spacing between contours is roughly 3 mm. The resulting average spacing in ψ is about double that in the XGC mesh.

2. The outer edge of the contour box is taken to be $R = 0.926$ m, large enough to yield continuous flux surfaces up to $\psi/\psi_{\rm sep} = 1.1$; the reason for doing so is described below.

3. The average poloidal spacing along the contours (the parameter `r_dtheta`) is 3 mm.

4. In `definegeometry2d`, the area of the triangles around the gas nozzle is set to 5×10^{-6} m^2, close to the area of a 3 mm equilateral triangle.

Given that the purpose of the synthetic GPI diagnostic is to generate data suitable for validating XGC, our approach is designed to use as much of the XGC simulation data as is possible. In the case of the application to Alcator C-Mod, this entails altering the way that the split limiter structure, which surrounds the GPI gas nozzles, is incorporated into the simulation. Given the relatively short mean free paths of the atoms and molecules in the vicinity of the plasma facing surfaces of the limiter, the precise shape of those surfaces is unlikely to impact the GPI light emission multiple centimeters away. Since those surfaces intersect the outer edge of the XGC mesh and since both are nearly flux surfaces, we opt to replace the limiter surface with a nearby flux surface; the specific one is noted below. The horizontal surfaces of the limiter and the gas nozzle are specified using engineering metrology data.

The steps described here summarize the original procedure. They would need to be modified for a new application.

In `efit2dg2d.web`, set the application macro:

```
@m APP CMOD_GPI
```

Then, compile and run the code with the EFIT equilibrium for the problem:

```
efit2dg2d XGC_GPI_19_ex/g1100223012.01149_261
```

As is noted in the preamble to `efit2dg2d`, the spline interpolation routine assumes that $\psi_{\rm sep} > \psi_{\rm axis}$. This may not necessarily be the case in general, especially for NSTX. Should the spline routine generate an error, the user should consult an EFIT expert for assistance in obtaining a file compatible with this assumption.

- This yields a set of contours in the resulting `wallfile`.

- The corresponding set of polygons is in the `polygons_dg2d.in` file.
- Both of these will be modified manually, as described in the next step.
- The `stratum_psi` file also produced relates the polygon stratum numbers to ψ and radial coordinates. This file is not used in this application since interpolation of plasma data is handled in a completely different way than in other `efit2dg2d` applications, such as ENDD.
- The `polygons.txt` file contains a point-by-point representation of the polygons in `polygons_dg2d.in` suitable for plotting; we will not be using this, either.

3.2.2 `definegeometry2d`

The construction of the 2-D input file to `definegeometry2d`, `xgc_gpi_19_dg2d.in`, begins with the preamble. The `symmetry` is `cylindrical`. The R-Z `bounds` values are 5 cm outside those of the box for `efit2dg2d`, except for outer radial boundary. The actual C-Mod vacuum vessel is at $R = 1.0616$ m. But, we set up these simulations with it at $R = 0.98$ m to reduce the volume of the vacuum region of the problem; this will be discussed in Sec. 3.4.3 in connection with importance sampling. However, still need to ensure that the camera vertex at $R = 1.025$ m is in the universal cell; we do this by setting the outer radius of `bounds` to 1.03 m. The `wallfile` keyword points to a modified version of the file produced by `efitdg2d`, as will be explained below. Finally, the preamble section of the input file is terminated with the `end_prep` keyword.

The next section of the input file consists of the polygons / strata, 1 through 34, extracted from `polygons_dg2d.in`. The polygons / strata 35 through 64 are deleted. The full set of polygons described in `xgc_gpi_19_dg2d.in` is depicted schematically in Fig. 3.1.

To construct the remaining polygons, we need to modify and extend the `wallfile` produced by `efit2dg2d`, resulting in the `wallfile.txt` pointed to by the `wallfile` keyword. Note that we do *not* delete the unused "walls" used in the construction of strata 35 through 64 because the last wall, wall 64, represents the complete outer boundary of the contour box; reconstructing it from the other walls would be possible, but extremely tedious.

We add four more walls to this file representing the upper and lower limiter structures, the gas nozzle, and the section of the vessel wall between the limiter structures. These point lists were created via a Python based script used in

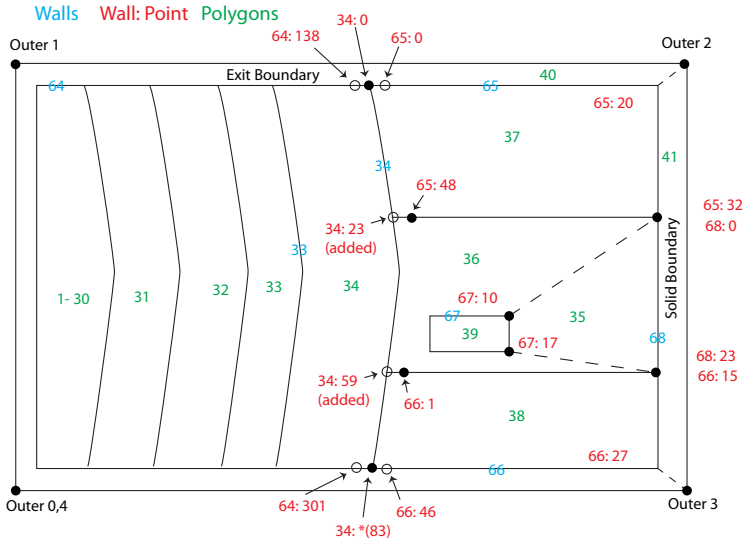

Figure 3.1: Schematic depiction of polygons in the `definegeometry2d` input file, `xgc_gpi_19_dg2d.in`. Polygon numbers are indicated in green; wall numbers in blue. Some key points are highlighted in red as (wall number):(point number).

constructing an earlier version of this geometry. Apart from their adherence to the C-Mod metrology data the key characteristic of these point lists is a roughly 5 mm spacing, consistent with the desired spatial resolution requirements listed above.

One other change is to add two points to wall 34 where it meets up with the limiter structures; the radial coordinates of these points is determined by manual interpolation. This change requires a corresponding increment to the number of points at the top of the file. Note also the insertion of the numbers of points in the four added walls.

The corresponding polygons are also manually constructed. The most important of these is stratum 36, labeled as "Remaining volume around nozzle", since it is closest to the region of interest (the GPI camera frame). As noted above, we set the triangle area of this volume to 5×10^{-6} m^2. We do likewise for the "Nozzle" polygon, stratum 39, since it contains plasma zones over most of the toroidal extent of the problem. The limiter halves and the volume behind the nozzle are less interesting, so we increase the triangle area to 5×10^{-5} m^2. Note also that all of the solid polygons are at the end of the file to allow the `ucd_plot` post-processor to work as intended.

Before running `definegeometry2d`, it must be compiled with the `detector_setup` routine, contained in `cmodgpi_camera4.web`, specifying the GPI and APD views. Note that the GPI camera is located at $\phi = 33^\circ$. From your run directory:

```
cp XGC_GPI_19_ex/cmodgpi_camera4.web ../src/usr2ddetector.web
gmake definegeometry2d
definegeometry2d XGC_GPI_19_ex/xgc_gpi_19_dg2d.in
```

The end result of this process, apart from the obvious geometry netCDF file, is a triangulation of the volume. To facilitate interpolation of the XGC data onto these triangles, we first extract them from the polygon netCDF file produced by `definegeometry2d`:

```
poly_to_tri XGC_GPI_19_ex/gpi_poly_2d.nc
```

The `poly_to_tri` routine produces the same sort of "node" and "element" files used by XGC. One difference is that the specification of the "element" file as defined in connection with the `Triangle` code allows for one or more floating point "attributes" to be associated with each element. We take advantage of that capability by adding a column at the end of the line containing the zone number associated with each triangle; this is formatted as a floating point number so as to be consistent with the `Triangle` code specification. In the 2-D case, the zone number is the same as the element number; this is not the case for 3-D.

Three Dimensional Geometry

The procedure for generating the 3-D geometry is very much the same. In the preamble of the `definegeometry2d` input file, `xgc_gpi_19_dg3d.in`, the `symmetry` becomes `cylindrical_section`, and we add the toroidal extent, going from -90° to 40°, to the `bounds` keyword.

The toroidal domain is divided much as in experimental GPI applications, with the resolution being greatest at the gas source and decreasing away from it. The toroidal range is chosen so as to encompass the full path of the camera view. The toroidal discretization is specified via the `y_values` command and is shown in Fig. 3.2; the XGC planes will be discussed in Sec. 3.3.1.

In the polygon construction section, the limiter sections and nozzle are solid only over a range of toroidal angles, as is indicated in `phi_values.txt` and depicted schematically in Fig. 3.2.

The toroidal boundary conditions are specified via the `y_min_zone` and `y_max_zone` keywords. We use `mirror` for both, even though this is not physical. While an exit boundary would be preferable here, that is not an option with these commands, which only require the user to specify a "material". Moreover, the recycling coefficient of that material is hard-coded to unity in `definegeometry2d`. Since the path of the camera chords extends in both directions well away from the gas source, we do not expect these boundary conditions to affect the results.

`definegeometry2d` does not need to be recompiled, assuming that `cmodgpi_camera4.web` remains as `usr2ddetector.web`:

```
definegeometry2d XGC_GPI_19_ex/xgc_gpi_19_dg3d.in
```

Note that this takes significantly longer to run than the 2-D case, perhaps 4 - 5 hours.

Even though the triangulation is the same as in 2-D, the associated zone numbers differ, as was noted above. Consequently, we label the polygon netCDF file with "3d" and, thus, the `Triangle` files as well:

```
poly_to_tri XGC_GPI_19_ex/gpi_poly_3d.nc
```

3.3 Plasma Setup

3.3.1 Interpolation of XGC Data in the Poloidal Plane

The first step in setting up the plasma background for DEGAS 2 is to interpolate the XGC plasma data at each of its toroidal planes from its triangular mesh to

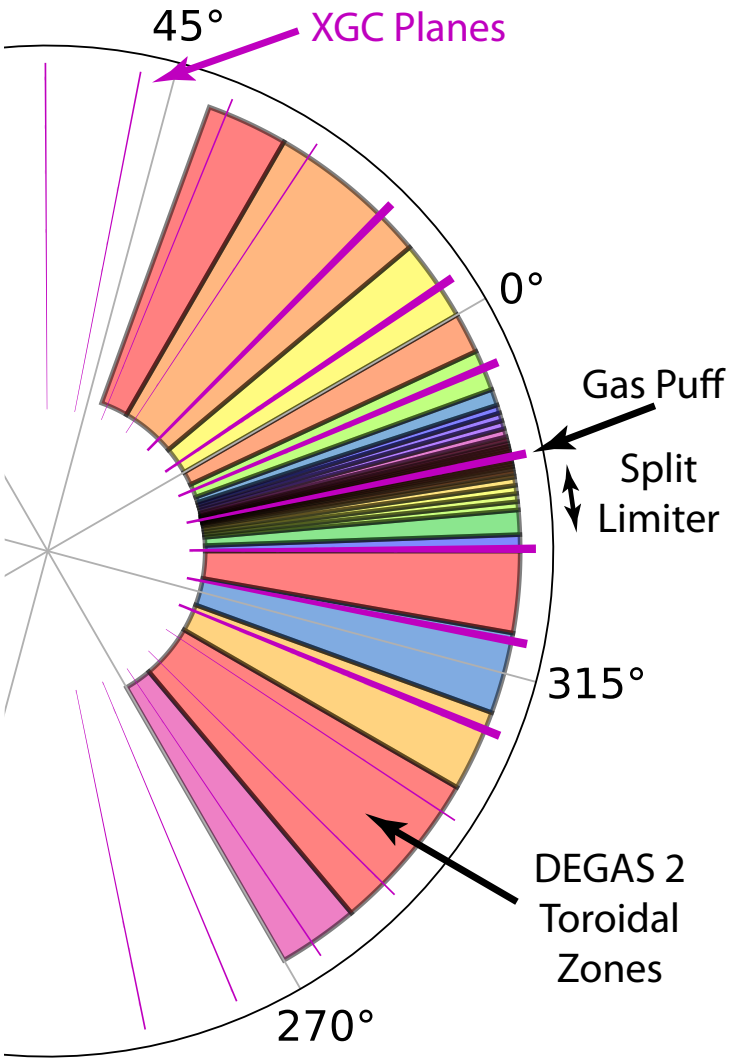

Figure 3.2: Schematic depiction of the toroidal discretizations in XGC and DEGAS 2. The magenta lines correspond to the 16 XGC toroidal planes; the thicker ones correspond to the planes used in the DEGAS 2 model. The colored sections represent the DEGAS 2 toroidal zones, corresponding to the values in `phi_values.txt`. The 10° slices at either end represent the toroidal boundaries for the problem; "mirror" boundary conditions are applied here. By design, the location of the gas nozzle is chosen to coincide with one of the XGC planes. The colors of the sections have no particular meaning.

the one constructed with `definegeometry2d`. This is done with the Python script `xgc1_gpi_plasma_v6.py`. The script requires four arguments on the command line:

1. The directory containing the `H5_files` directory, which in turn has the XGC plasma files. Since there may be hundreds of these files, placing them in a subdirectory is convenient.

2. The name of the XGC plasma file to be processed; this represents a single XGC time slice. Note that these are XGC1's `f3d` output files and are produced in `binpack` format with a `.bp` extension. To simplify the task of working with them in Python, they have been converted to HDF5 format using the `bp2h5` utility. The integer between the `f3d` and `h5` in the file name is the XGC output file number.

   - In this run, the XGC variables `sml_dt` = 9.4×10^{-8} s and `diag_3d_period` = 2 (by default this is equal to the input parameter `diag_1d_period`) so that each file corresponds to 1.88×10^{-7} s.
   - This run, `ti267_cmod_IP_08`, had 484 output files, corresponding to 968 total XGC steps.
   - Since the plasma variations over the first 100 files are small, the synthetic diagnostic is started at file number 101, with file 100 used for initializing the neutral distribution, see Sec. XX.

3. The directory and root file name for the target triangulation onto which the XGC data will be interpolated; the script will add `.ele` and `.node` to the root file name.

4. The directory to be used for the output text files.

For this example,

```
XGC_GPI_19_ex/xgc1_gpi_plasma_v6.py XGC_GPI_19_ex/ xgc.f3d.00100.h5
         XGC_GPI_19_ex/gpi_poly_2d XGC_GPI_19_ex/
```

The script loads the XGC's full triangular mesh via the `xgc.py` module (provided as part of this example). The target triangulation for DEGAS 2 is then read in from the `.ele` and `.node` files. Python and NumPy tools are then used to set up the interpolation of the node-based XGC data onto the triangle centers, as

is required by DEGAS 2 (all quantities are constant over a triangle / zone). This yields a set of "barycentric coordinates" that can then be used to perform the interpolation from the XGC mesh to DEGAS 2 mesh in a single expression.

The script reads in the requisite XGC plasma data from the file and then carries out the interpolations with these adaptations:

- The ion density is set equal to the electron density.
- The minimum density is 10^{15} m^{-3}.
- For both the electrons and the ions, an average temperature is computed via:

$$T = \frac{2}{3}(E_{\parallel} + T_{\perp} - \frac{1}{2}mv_{\parallel}^2), \tag{3.1}$$

  where $E_{\parallel}$ for electrons is `e_E_para` in the `f3d` file, likewise `e_T_perp` $\rightarrow T_{\perp}$ and `e_u_para` $\rightarrow v_{\parallel}$. Analogous associations are made for ions.
- The minimum temperature is 1 eV.
- For the ions, the parallel velocity is also written out for use in the DEGAS 2 simulations.
- To get axisymmetric values for all plasma parameters, we average over all toroidal planes.

For this XGC run (and all of the others considered in the development of the synthetic diagnostic), `sml_wedge_n` = 2 so that the 16 toroidal planes in the file occupy only half of the torus and each toroidal division is $11.25°$ wide; see Fig. 3.2. Only seven of the XGC planes are needed to span the DEGAS 2 zones crossed by the camera view. The mid-point of these planes is aligned with the gas nozzle at $\phi = 18.48°$. Arbitrarily, we take XGC plane number five as the first of the seven planes, going clockwise in the figure. Separate files are written out for each toroidal plane, with the plane number incorporated into the file name, e.g., `plasma_file_3.txt`. The plasma data are written out in `defineback`'s "Tabular Format". In principle, other formats can be used if appropriate modifications are made to the `get_n_t` routine.

An eighth file, `plasma_file_axi.txt`, contains the axisymmetric values in the same format. A key difference between this and the other files is that the minimum density and temperature are reduced to zero. Since negative axisymmetric plasma data are extremely unlikely, we do not foresee these minima being

enforced in DEGAS 2 triangles that overlap XGC nodes. Consequently, we will subsequently use the appearance of zero values in this file to identify triangles that are outside the XGC mesh. Some other more direct mechanism for labeling these triangles, e.g., in a separate file, could in principle be developed should the need arise.

3.3.2 Field Line Interpolation

Were the DEGAS 2 and XGC toroidal discretizations identical, constructing the DEGAS 2 plasma background from the files written in the previous step would be trivial. However, the requirements of the algorithms in the two codes are fundamentally different and, thus, so are the toroidal grids. To map between the two, we interpolate the data along a field line. The fundamental physical principle is that the variations of the plasma parameters along a field line should be much smaller than those in the directions perpendicular to it.

Initial treatments did exactly this. However, in one NSTX simulation, the XGC data exhibited greater density variations along a flux surface than along a field line, invalidating the simple procedure. The solution is to instead interpolate the difference between the local values and the axisymmetric one, then add the axisymmetric value back.

The fundamental equation used for field line following is:

$$\frac{d\ell}{B} = \frac{dx}{B_x} = \frac{dy}{B_y} = \frac{dz}{B_z}. \tag{3.2}$$

Or, in cylindrical coordinates:

$$\frac{dR}{B_R} = \frac{dZ}{B_Z} = \frac{Rd\phi}{B_\phi}. \tag{3.3}$$

This leads us to the actual expressions for integration along a field line:

$$\frac{dR}{d\phi} = R\frac{B_R}{B_\phi}, \tag{3.4}$$

$$\frac{dZ}{d\phi} = R\frac{B_Z}{B_\phi}. \tag{3.5}$$

The magnetic field line structure is derived from the EFIT equilibrium. First, the poloidal flux and other data are fit with spline functions by subroutine `init_psi_interp`

in `psiinterp.web`. This routine calls subroutine `test_interp` upon completion to compare spline fits with the actual values of the equilibrium quantities. Errors in the spline fits are computed, although no feedback is provided to the user. These should be examined in cases exhibiting unexpected behavior. The spline fits form the basis of subroutine `bvec_interp`, which returns the magnetic field vector at the input R and Z.

Subroutine `follow_field`, also in `psiinterp.web`, uses Eqns. (3.4) and (3.5) to take a small step along a field line through an input range of toroidal angle, $\Delta\phi$. The subroutine uses a Runge-Kutta scheme of first, second or fourth order, as determined from its first argument. For this application, second order is used; this is specified via the `rk_order` macro in the preamble of `xgc1_gpi.web`. A detailed convergence test was performed for all three orders, examining the scaling of the errors with the step size. The first and second order integrations scaled as expected; the fourth order did not. Integrations over a long distance, 10 times around the torus, confirmed these findings.

Note that the experimental GPI systems are designed for a particular orientation of the magnetic field line, as well as being optimized for a specific field line pitch. Since the geometry of these simulations mimic that experimental configuration, the user needs to ensure that field line direction used in the XGC run is the same as in the experiment, or at least anti-parallel to it.

The relative arrangements of the XGC toroidal planes, the DEGAS 2 toroidal zone centers, and the boundaries of the problem, both toroidally and vertically, lead to three different situations, as is depicted in Fig. 3.3.

1. Integration in both directions remains within the bounds of XGC mesh subset, yielding two XGC data values and distances to both $\Rightarrow$ we can legitimately interpolate between the two.

2. Integration remains within the box in only one direction. Given the notion that plasma parameters, including those in "blobs", should be roughly constant along field lines and that our interpolation distances short compared with the parallel connection length, we just use the plasma value that we have and use it directly.

3. Integration misses the XGC planes in both directions. We use the axisymmetric value in this case.

Once the geometry has been set up, the required interpolation factors for each DEGAS 2 zone can be determined and will not change. Hence, we created the

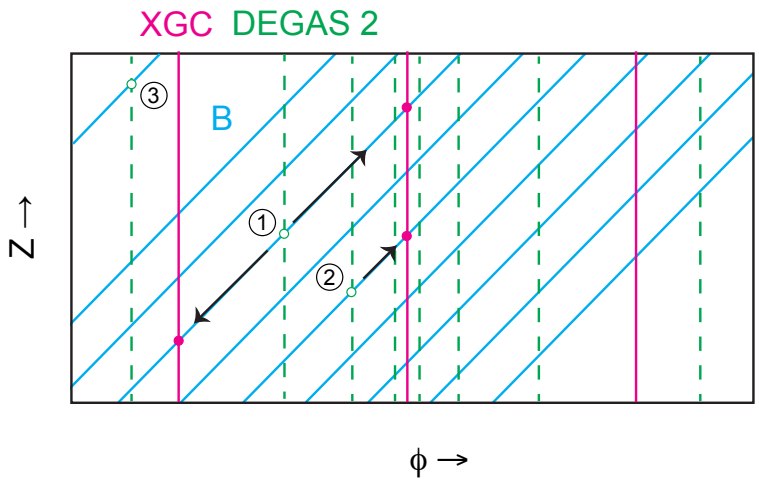

Figure 3.3: Schematic depiction the field line interpolation scheme. The magenta lines represent the XGC toroidal planes. The dashed green lines represent the centers of the DEGAS 2 toroidal divisions. The blue lines represent the local magnetic field lines.

XGC interpolation class, `xgc1interp.hweb`, to hold that information in a netCDF file; the constituent arrays are documented at the end of the file. Once this file has been created, it can be read in when processing subsequent time steps, saving considerable time.

3.3.3 `defineback`

The `get_n_t` subroutine for the synthetic GPI example is in `xgc1_gpi.hweb` (included with the DEGAS 2 distribution), which should be copied to `usr2dplasma.web` and compiled into `defineback`. Note that this routine relies on the "XGC1 interpolation" class (`xgc1interp.hweb` and `xi_common` in the code). Since this is presently the only invocation of this class in DEGAS 2 and since the `Makefile.depends` file generally does not contain entries for the (problem dependent) `usr2dplasma.web` file, the user should execute `gmake depend` in the `src` directory before compiling `defineback`. As with virtually all custom `get_n_t` subroutines, the argument to the `plasma_file` is a string. In this case, it consists of the paths to three files:

1. The list of plasma files generated by the Python script, as described above in

Sec. 3.3.1. The format of this list is prescribed at the top of `xgc1_gpi.web`.

2. The EFIT g-file; this should be the same one used in Sec. 3.2.1.

3. The netCDF interpolation file. If this file does not yet exist, the interpolation factors will be computed along the way and the file written out. If it does exist, the factors will be read in from the file. For axisymmetric simulations (the number of files specified on the first line of the plasma file list is 0), these steps are skipped completely, although the subroutine still expects to find a string here.

We use generic file names for the plasma file coming from the Python script so that they will be overwritten with each time step. Consequently, the `plasma_file` argument also remains the same in going from one XGC time step to another. In fact, the only argument in the input file that varies is `time_interval`.

3.4 Three Approaches to Time Variation

The contents of the remainder of the input file to `defineback` depend on the approach used for handling the time variation in the XGC data. To simplify progress through this example, we will present them in order of increasing complexity. The designations of the methods is that used in the synthetic GPI paper.

3.4.1 Frozen Plasma Fluctuations (FPF)

A separate, steady state run is performed on each XGC time slice; this assumes that the neutral transport time scales are much faster than those of the plasma turbulence. This mode of operation is also the most like that used in experimental GPI simulations. Since the DEGAS 2 runs are all steady state, the `time_interval` keyword is not needed in the `defineback` input file, so the same file, `xgc_gpi_19_db_ss` (`xgc_gpi_19_db_3D_ss` for 3-D) can be used for each time slice. Note that the desired number of flights is specified here so as to avoid having to manually set the number in the background netCDF file.

The baseline input file for `tallysetup`, `tally_xgc_gpi_v2.input` is used; it contains tallies for the avalanche photo-diode arrays (APD), but not the GPI image. No other special considerations are needed for `flighttest` or `outputbrowser`.

We do need to set the appropriate macro switches in `postdetector`. Edit the lines:

```
@m APP NSTX

@#if 0
```

to read:

```
@m APP CMOD_MID

@#if 1
```

The first enables device specific macro settings later on, e.g., for specifying the target plane. The latter toggles between the full camera view and a subset used for testing. Note that with the implementation of compression of the `zone_frags` array, the code runs quickly even with a full frame. The same `detector_setup` routine compiled into `definegeometry2d`, `cmodgpi_camera4.web`, is still needed as `usr2ddetector.web`.

The 2-D FPF simulation is sufficiently quick and simple that we suggest running it as an interactive batch job with the sample batch script, `gpi_script_fpf_int`. A few details to note:

- The parallelized `flighttest` executable has been renamed `flighttest_mpi` to avoid successive recompiling to switch between scalar and parallel versions.
- The script assumes that the `Camera_FPF` directory already exists.
- The short file `a_no` serves two purposes in this script. First, it provides the "no" response to force `outputbrowser` to exit its interactive mode following its execution of the commands in its input file. Second, the prior existence of this file effectively aborts the batch job. As is pointed out in the script's comments, doing so prevents uncontrolled restarts of the batch job, which can overwrite "good" output files with problematic ones. Be sure to delete `a_no` before running this script.

For the purposes of these examples, we use XGC files 301 – 313 since they contain a relatively large scale instability that can be easily seen in the GPI images; we will discuss this further in Sec. XX on visualization. The GPI image for each step

is generated by `postdetector` and stored in the `Camera_FPF` directory, as are the APD signals. Note that little use of the latter has been made to date. Thirteen steps are used to demonstrate the integration over multiple steps described in the paper to improve statistics (for the time dependent approach) and to achieve an effective frame rate comparable to that of the GPI camera. The resulting image is similar to that of the individual frames. The `acc_output` code used for the integration takes advantage of the same machinery used for restarts to combine the contents of one DEGAS 2 output netCDF file with another one from the same problem; see the premable to `acc_output` for additional details.

Once the proper operation of the executables involved in the batch script has been established, the executables `defineback`, `outputbrowser`, `postdetector`, and `acc_output` can be compiled with `DEBUG = no`. For `flighttest`, parallelization should be turned on with `MPI = yes` and the `MASTER_SAMPLE` sample switch should be left with the default `no`. The two exceptions to these guidelines are that `flighttest` must be run in scalar mode for the Fixed Neutral Density approach, Sec. 3.4.2, and that `MASTER_SAMPLE` should be `yes` for `flighttest` in the time dependent case, Sec. 3.4.3.

3.4.2 Fixed Neutral Density (FND)

A single steady state run is performed on a representative, axisymmetric background plasma to obtain a 3-D neutral background that will be held fixed in time. With that in hand, the XGC plasma data for each time step are read in via `defineback`, just as in the FPF mode. The principal difference is that `flighttest` is run in restart mode with the same number of flights as in the initial neutral background run. Since no additional flights are requested, the code's primary loops are bypassed. In this case, however, all of the light emitting reactions are set up to be computed via the post-processing estimator. The emission rates are then appropriate to that particular time step, but the neutral density is that corresponding to the initial run. These runs should be performed with `flighttest` in serial mode, compiled with `MPI = no`. The time stepping portion of this method is sufficiently quick that working in 3-D is practical even for an example calculation; the procedure for generating it was outlined in Sec. 3.2.2. See also the dedicated input file, `degas2_FND.in`.

In the synthetic GPI paper, the plasma used in producing the fixed neutral background was taken from a time step in the middle of the XGC run; this matters since the plasma profiles evolve during the XGC simulation. That is not the case for this much shorter example, so we simply use the first step from the set that we

will be working with:

```
XGC_GPI_19_ex/xgc1_gpi_plasma_v6.py XGC_GPI_19_ex/ xgc.f3d.00301.h5
XGC_GPI_19_ex/gpi_poly_3d XGC_GPI_19_ex/
```

As described in Sec. 3.3.1, this writes to the output directory (`XGC_GPI_19_ex` here) a separate text file for each XGC poloidal plane, plus one with the axisymmetric values. To avoid confusion between the "neutral background" and the plasma background, we will henceforth refer to the former as "NB".

We only need the axisymmetric file in producing the NB, so this initial `defineback` input file points to `plasma_file_list_2d.txt`. It also includes `source_segment_iy` keyword required for 3-D:

```
defineback XGC_GPI_19_ex/xgc_gpi_19_db_FND_init
```

We specify here the target number of flights, 8×10^6, since manually editing the 3-D background netCDF file is not all that easy.

The same input (and output) file for `tallysetup` is used in both the initial NB run and the subsequent restart steps. The most relevant difference between this input file and the ones used with the other synthetic GPI time stepping methods is that the post-processing estimator is selected for the `ionize`, `ionize suppress`, `dissociation`, and `test ion` reaction groups (see the documentation for tallysetup). Doing so violates the general requirement of using the collision estimator for the "test ion" group, causing an assertion to be thrown:

```
Assertion failed (tallysetup.web):
    ' Test ions must use collision estimator'=='
```

The D_2^+ densities are properly computed during the main loop of `flighttest` so that using the post-processing estimator to then determine the associated light emission is not a problem. The user can either type "go" in response to the assertion (four times), or compile `tallysetup` with `DEBUG = no`, bypassing the assertion machinery.

The full, 3-D run of `flighttest` with 8×10^6 takes about an hour with $\sim 10^2$ cores. Once complete, run `postdetector` to generate a corresponding camera image; we will need this for normalizing the actual time dependent images. In a typical application, as in the synthetic GPI paper, one would instead normalize to the initial, quiescent time slices from XGC. Since these time slices are far from quiescent and very similar to each other, normalizing to the first of them would remove much of the structure we wish to see.

Rename the output file so that it is not accidentally overwritten:

```
mv degas2_xgc1_cmod_out.nc degas2_xgc1_cmod_NB.nc
```

Likewise, copy the `postdetector` file to the output directory to be used for this run (create it if it does not already exist):

```
mv degas2_xgc1_cmod_out_post.nc
Camera_FND/degas2_xgc1_cmod_out_post_NB.nc
```

As in the FPF run, the batch job is sufficiently quick that it can be run interactively, this time on a single processor, via the script `gpi_script_fpf_int`. Some details:

- Again the file `a_no` is used to prevent unwanted restarts and must be deleted before starting the script.
- The initial call to `defineback` will take longer than subsequent ones since the interpolation file (`xgc1_int_19.nc`; see Sec. 3.3.2) will be generated (the run for the NB was on an axisymmetric plasma).
- The same set of XGC files, 301 – 313, is used.
- This same `defineback` input file, `xgc_gpi_19_db_3D_ss` could be used to perform a 3-D version of the FPF run.
- All of these invocations of `flighttest` are "restarts" with the same number of flights as in the NB run. Consequently, the flag `so_restart` in the background netCDF file needs to be changed from its default value of 0 to 1. We do this via a few editing commands since `defineback` offers no way to do this via its input file.
- These editing commands could in principle be collapsed into a single command via "pipes", but have been separated to minimize the likelihood of errors.
- The `postdetector` output files are copied to the `Camera_FND` directory.

Note that the `acc_output` code is *not* used in this application to integrate over frames since the statistical uncertainty in each frame is exactly the same. We instead perform simple averages over frames in the visualization stage, Sec. XX.

3.4.3 Time Dependent Neutral Density (TDND)

We will initially go through this in 2-D, which allows us use the same input file as the FPF mode:

```
cp XGC_GPI_19_ex/degas2_FPF.in degas2.in
```

Since we already have the geometry file, we can proceed to set up the snapshot distribution corresponding to the initial time. First, interpolate the XGC data onto the DEGAS 2 triangles:

```
XGC_GPI_19_ex/xgc1_gpi_plasma_v6.py XGC_GPI_19_ex/
    xgc.f3d.00301.h5 XGC_GPI_19_ex/gpi_poly_2d XGC_GPI_19_ex/
```

Then, run `defineback` with the dedicated input file for the initialization run:

```
defineback XGC_GPI_19_ex/xgc_gpi_19_db_init
```

Note that we are using the 2-D (axisymmetric) plasma file here. The end of the time interval, 1.8802×10^{-7} s, will become t_0 in the subsequent time dependent run. The critical keyword in this input file is `time_initialization`; this sets the `so_time_initialization` flag in the sources class and in the "Time Dependence" section of the User's Manual. The number of flights needed to adequately populate the snapshot distribution has already been set in this input file.

Again, because this run uses the same geometry (and `problemsetup` input file) as the FPF part of the example, the `tallysetup` netCDF file is the same. No harm can come from rerunning `tallysetup` at this point, however.

Next, the user need only run `flighttest` to create the snapshot distribution. To prevent the snapshot file from being accidently overwritten, it has been copied to `sn_xgc1_cmod_init.nc`. Should the user want to restart the time dependent run from the beginning, it can be copied back to the `sn_xgc1_cmod.nc` pointed to by the `degas2.in` file. The output netCDF file from this initialization run can be used to check the statistics, etc., but is otherwise not needed for the rest of the run.

Improving Statistics

Before beginning the actual time dependent run, we need to examine the associated statistics, which differ dramatically from those of steady state simulations.

First note that the geometry of these runs has a substantial plasma-free volume behind the gas nozzle. This region is dominated by slow moving molecules or atoms that do not contribute to the GPI signal. The principal reason for including it in the simulations is to provide a physically realistic boundary condition for atoms returning from the plasma following a charge exchange or dissociation event. A first step in mitigating this problem is to reduce the major radius of the vessel wall from $R = 1.0616$ m to $R = 0.98$ m, as was noted in Sec. 3.2.2; the remaining 5 cm of space behind the gas source is more than enough to provide the desired boundary condition. For example, a 3 eV atom (300 K D_2 molecule or He atom) needs 18 (267) time steps to traverse it.

Another confounding factor is the short mean free path for atoms heading into the plasma, dropping from an effectively infinite 25 cm near the nozzle to between 1 and 3 cm around the separatrix. Since the separatrix is 3.7 cm from the nozzle in this shot (at the vertical center of the GPI frame), the atoms must travel more than one mean free path to reach the confined plasma. But, we need even deeper penetration than this to reach the regions in which the plasma turbulence is most active.

Two techniques are used to improve the statistics in the region of interest, i.e., the GPI "target plane." The first, "suppressed absorption," (also called "survival biasing") is commonly used in DEGAS 2 in connection with the atomic ionization process. When invoked, the atoms do not undergo discrete ionization events in which they are completely ionized in a single collision. Rather, the statistical weight ($= 1$ when the particle is sampled) of the atom is exponentially reduced along its track with a rate equal to the ionization rate. To avoid the tracking of statistically insignificant particles, a minimum statistical weight is set. Once that weight is reached, the decision to either terminate or continue the particles track is made randomly with a 50% chance for each ("Russian roulette"). A typical steady state DEGAS 2 GPI simulation employs a minimum weight of 10^{-3}. In the present case, however, very small weights will be the norm. To ensure that flights can populate the inner region as fully as possible, we reduce the minimum weight to 10^{-7}. The impact on the run time is modest. To effect this, the user needs to set the parameter `WMIN` in `flight.web`:

```
@m WMIN const(1.,-7)
```

Be sure to return this to the default of 10^{-3} for use with more standard problems.

The second technique, importance sampling, is critical for making the optimal use of the snapshot distribution. By default, the probability for sampling a particular particle from the snapshot distribution is proportional to its statistical weight.

Because the cold molecules and atoms in the vacuum region have undergone little or no ionization, their statistical weight is comparable to that of newly sampled gas puff particles. Moreover, the neutral density there is two or more orders of magnitude larger than in the GPI target region. Consequently, a direct sample of the snapshot distribution would be dominated by the cold molecules and atoms in the vacuum region. Moreover, these particles represent a simple distribution (thermal molecules or atoms) that can be described adequately with relatively few particles. In contrast, the particles in which we are most interested comprise the very kinetic distribution of atoms resulting from the charge exchange reactions and perhaps containing complex spatial variations associated with the plasma turbulence.

The weighting scheme we employ in the importance sampling is designed to yield a snapshot source consisting of a radially uniform number of particles. First, we bin all of the snapshot particles by major radius in 1 cm wide bins. The inverse of this distribution then provides the relative weight adjustment factors for the particles in each bin; these correspond to the $r_{i,l}$ in Sec. 2.5 earlier in this manual and are computed in subroutine `set_snapshot_source`. For example, the less-common and lower weight particles well inside the separatrix will be more heavily weighted, increasing the likelihood that they will be sampled. The obvious bias that this introduces is compensated by then decreasing the statistical weight carried by these particles during their tracks, via Eq. (2.33). The opposite is true of snapshot particles that are sampled at larger major radius.

To enable this, the user needs to set the macro flag in `writeback.web`:

```
@m GPI_WEIGHTING 1  // Set to 1 for synthetic GPI weighting
```

This macro also contains code that writes out this binned distribution to a file called `snapshot_rel_wts` for monitoring after the run. Again, since this code is specific for this geometry, its use in other applications may cause problems; be sure to reset the macro to its default value of 0. Once `defineback` is compiled with this enabled, the snapshot source will be assigned these weighting factors. The file `doflights.web` has a corresponding macro that writes out the resulting distribution of sampled particles to `snapshot_samples` that can be used to illustrate the effect of importance sampling.

Figures 3.4 and 3.5 reflect the snapshot distribution from the initialization run, as written to `snapshot_rel_wts`, and the subsequent sampling of it, from `snapshot_samples`, when `flighttest` ran the actual first time step. The notebook used to generate these plots, `XGC_GPI_19_ex_snapshot_samples.ipynb`, is included with the example files.

The left plot in Fig. 3.4 is nothing more than a pictorial representation of the sampling process itself; alternatively, it is a plot of the spatial distribution of the overall neutral probability distribution function. Both curves are based on the total weight in each bin, regardless of species. The conclusion here is that the relative weighting and sampling are both working as intended.

The right plot shows the impact of the relative weighting on the number of particles (regardless of weight) in each bin; keep in mind that the key to reducing the variance is to increase the number of scores, i.e., particles, assuming all of the particles are of similar weight. The red curves depict the problematic situation that arises with weights fixed at 1: most of the particles are D_2, and the bulk of these are in an uninteresting region. The blue curves appear to, and probably do, sum to roughly a constant; this was the intent of the relative weight distribution. The result is a huge increase in the number of D in the plasma volume, with no change in total weight, and a reduction in the number of D_2 in the vacuum region. Note that we add "1" to the D_2 curves to allow plotting on a log scale since their number goes to zero at small R.

Figure 3.5 is analogous to the left plot in Fig. 3.4, but broken down by species. This demonstrates that the sampling procedure separately preserves the weight distribution for each species, even though no explicit provision for doing so has been made.

Execution

Due to the large amount of memory occupied by the snapshot distribution and the associated source, the user should always perform time dependent runs of `flighttest` compiled with `MASTER_SAMPLE = yes` in the `Makefile.local` file, even in 2-D. This will dramatically reduce the time required for the initial MPI broadcast to the slaves, as well as the memory they consume.

The batch script for this run, `gpi_script_tdnd`, differs from the ones for the FPF and FND methods in that it:

- Is intended to be run purely as a batch job, i.e, not interactive.
  - This is not all that critical for this example, which requires about two hours to run.
  - However, 3-D runs require additional care in setting up the processors; see below.
  - And can take days or even weeks to run the full set of XGC time steps.

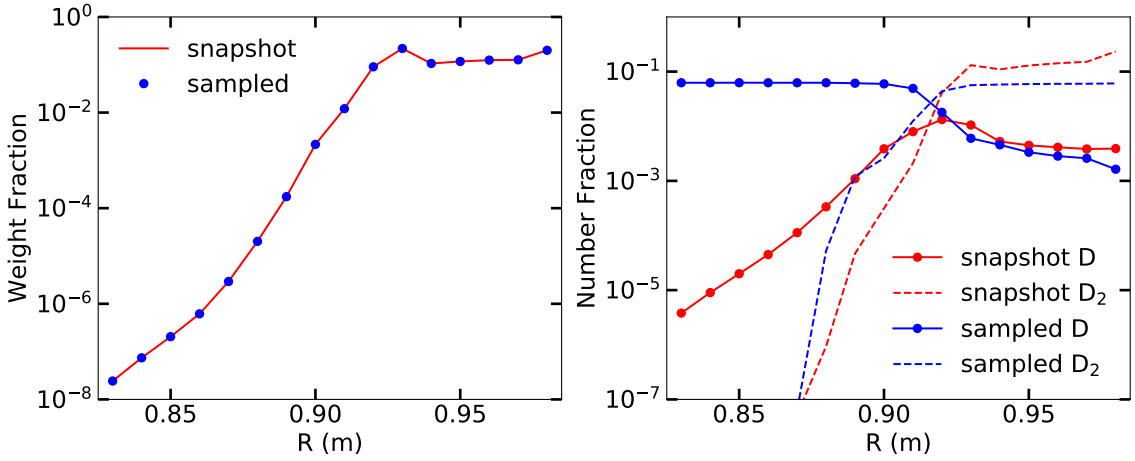

Figure 3.4: Binned probability distribution functions based on particle weight (left) and particle number (right). Both plots contain lines for the snapshot distribution, as contained in the snapshot netCDF file, and for the distribution of particles sampled from it prior to tracking in the main code. The decoupling of particle number and weight apparent in the "sampled" plots is the objective of importance sampling.

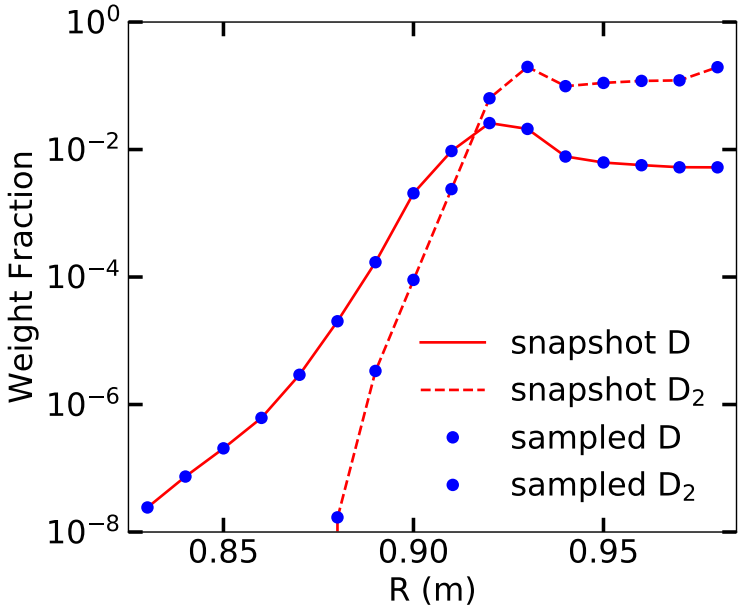

Figure 3.5: Binned probability distribution functions based on particle weight, separately for D atoms and D_2 molecules.

- Keeps track of the physical time step, needed for the `defineback` input file.
  - The `T_ZERO` set at the top of the script is the starting time of the run, representing the end of the initialization time step.
  - Consequently, `T_END` is initialized to this same value; at this point, this is also the time associated with the snapshot particles from the initialization run.
  - At the beginning of a time step, `T_START` is set to `T_END`,
  - Then `T_END` is computed from `T_ZERO`, the time step size, `DT`, and the step number, `INT_STEP`, which is read from the file name.
  - Note that in this case, we subtract 300 from the step number since the first file is for step number 301.
  - The values of `T_START` and `T_END` are written into the `defineback` input file via a `sed` command which operates on a "base" version of the input file, `xgc_gpi_19_db_base`.
- Compiles diagnostic information on the snapshot distribution and on the sampling thereof.
  - These files are as described above in Sec. 3.4.3.
  - But, are concatenated into a single file of the same name in the `Camera_TDND` directory with the step number separating each section of the output.
- The number of particles in each snapshot file is compiled in the file `snap_flights.txt` in the `Camera_TDND` directory.
- As in the FPF run, the final section of the script integrates the output over 13 time steps using `base_xgc1_cmod_out.nc` to contain the intermediate output.
  - One difference is that we also store every thirteenth snapshot file, labeled with the step number, in the `Camera_TDND` directory as a checkpoint and for post-run analysis.

Restarting

When running hundreds of XGC time steps in total, one may need to break them up into multiple batch jobs to stay within the constraints of the batch system. The best point to stop a job is at the end of a "thirteenth" step since the script will automatically have saved a copy of the last snapshot file, reducing the likelihood of it being deleted or overwritten. Otherwise, one just needs to ensure that the snapshot and base output netCDF files from the end of the last time step are valid. The other things one needs to do:

- Set the step number.
  - Namely, modify the list of XGC files so that it begins with the next step. E.g., do something like:

    ```
    XGC1filelist=`ls -1 $XGC1dir/H5_files/xgc.f3d* | tail -163`
    ```

    where the "163" was chosen to yield the desired step.
- Set `T_END` to the time associated with the particles in the snapshot netCDF file.
- Delete the `a_no` file, as well as the various standard output and error files for the batch script and the various executables.

Should the job be interrupted in the middle of a step, the user will need to examine the various files and make whatever adjustments are needed to allow the batch job to be restarted from the beginning of that step.

Differences in 3-D

The memory and storage requirements in 3-D are, not surprisingly, greater than in 2-D. The use of the `MASTER_SAMPLE` option minimizes the load on the slave processes. In the original set of TDND runs, extra steps were taken to ensure that the master process had access to adequate memory. First, the script `gen_host_file` (included in the example directory) was developed to provide detailed control over the cores on the master node. In particular, this script removes 5 cores from the allocated list on that node so that the memory associated with those cores can be utilized by the master process. Accordingly, the variable `NPROCS` in the batch script needs to be 5 smaller than the number of cores requested by the `SBATCH` command at the top of the script. The name of the file created by `gen_host_file` is passed on to `mpirun` via its `hostfile` argument. E.g.,

```
HOST_FILE_NAME=`gen_host_file`
ARGS="--hostfile "$HOST_FILE_NAME
```

The batch script `gpi_script_20` illustrates the practical application of this tactic. The user will need to adapt the details according to the size of the run to be performed and available resources.

3.5 Visualization and Post-Processing

The first and simplest visualization of the camera frames produced by post-detector is demonstrated in the three Jupyter notebooks included in the example, one for each approach: `XGC_GPI_19_ex_FPF.ipynb`, etc. These include overlaid surfaces representing the separatrix and XGC's outer boundary. The movies can be written to an mp4 file by using the `save` command of the `matplotlib.animation` package. Similar steps can be taken to generate an analogous movie from the XGC output files, as in the synthetic GPI paper.

The quantities used to characterize the turbulence, auto-correlation time, correlation lengths, etc., should be computed from these frames using the exact same methods that are applied to the experimental data. Doing so ensures an "apples-to-apples" comparison.

Contents